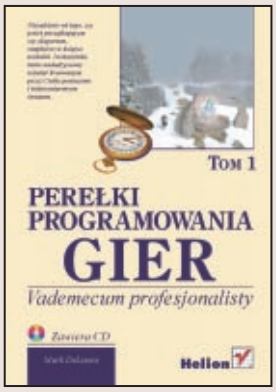
Perełki programowania gier. Vademecum profesjonalisty. Tom 1



Autor: Mark DeLoura
Tłumaczenie: Rafał Jońca
ISBN: 83-7197-704-2
Tytuł oryginału: Game Programming Gems
Format: B5, stron: 638

W niniejszej książce znajdziesz połączoną wiedzę ponad 40 utalentowanych twórców gier. Współpracując, stworzyli zbiór wskazówek dotyczących programowania gier, dzięki któremu uzupełnisz swoją wiedzę. Jeśli zaimplementujesz zaprezentowane tutaj techniki (wypracowywane przez wiele godzin), wrogowie będą sprytniejsi, bohater płynnie powali przeciwników, a gracze z powodu wysoce realistycznego trójwymiarowego świata będą się bali zgasić światło w nocy.

Niezależnie od tego, czy pytali mnie o nowinki techniczne wprowadzone w nowej konsoli, czy o złożone algorytmy, jedna rzecz stawała się dla mnie jasna: ciągle zadajemy pytania. Jako programiści gier często nie wiemy, jak wykonać postawione przed nami zadanie. Może właśnie dlatego tak bardzo lubimy tę pracę! Co masz jednak zrobić, gdy przytrafi Ci się opisana sytuacja? Przeszukasz domową biblioteczkę lub zasoby sieci WWW? A może zajrzysz do archiwalnych numerów fachowych czasopism? Żaden twórca gier nie korzysta z jednego określonego źródła. Czy nie byłoby wspaniale, gdyby jednak istniało takie miejsce, do którego zawsze zajrzysz w pierwszej kolejności? Właśnie w tym celu napisaliśmy książkę, którą trzymasz w ręce.

Rozdziały książki obejmują wiele problemów technicznych, na które możesz się natknąć, pisząc grę. Znajdziesz ogromną liczbę szczegółowo omówionych technik, ale i kilka bardziej ogólnych rozdziałów. Zadaniem książki jest zwiększenie Twojego stopnia zaawansowania niezależnie od aktualnej wiedzy, jaką posiadasz. Na przykład w bardziej ogólnych rozdziałach opisujemy techniki, nie zagłębiając się w szczegóły; na ich omówienie czas przychodzi później. Dobrymi przykładami mogą być rozdziały o kwaternionach oraz część dotycząca algorytmów sztucznej inteligencji.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Rozdział 4.5

Wykrywanie zderzeń w trójwymiarowej przestrzeni

Kevin Kaiser

Mechanizm fizyki działający w czasie rzeczywistym jest najważniejszym elementem, dzięki któremu można tworzyć trójwymiarowe środowiska, na widok których gracz po prostu kręci głową z niedowierzaniem. Mechanizm symulacji fizyki zapewnia realistyczną interakcję obiektów. Wtedy gracz czuje realizm przedstawionego świata, a co za tym idzie, może lepiej się po nim poruszać, ponieważ zachowuje się podobnie do tego, do czego jest przyzwyczajony w realnym świecie. Pierwszym i najważniejszym krokiem przy tworzeniu realistycznej symulacji fizyki jest dokładne wykrywanie zderzeń; gdy już zostanie jakieś wykryte, symulacja może odpowiednio zadziałać. W tym rozdziale ułożymy podwaliny pod tworzenie perfekcyjnej symulacji fizyki, omawiając najważniejszy jej fragment, czyli wykrywanie zderzeń w trójwymiarowej przestrzeni.

Algorytmy

W rozdziale zajmiemy się dwoma algorytmami detekcji zderzeń:

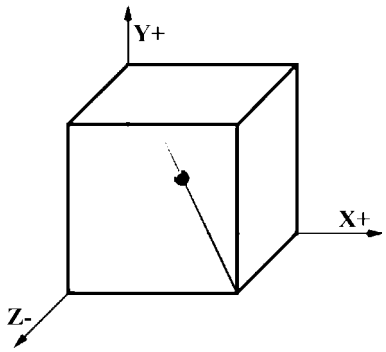
- **Wykrywanie zderzeń na podstawie kul otaczających obiekty.** Używamy ich, ponieważ kod jest stosunkowo prosty, a wyjaśnienie zasad działania nie nastręcza większych problemów. W zasadzie kod sprawdza zderzenia, testując promień jednej kuli z promieniem drugiej.
- **Wykrywanie zderzeń na podstawie przecięć trójkątów.** W tym przypadku, zanim przejdziemy do algorytmu, warto będzie sobie przypomnieć co nieco z matematyki. Ten algorytm używa równań parametrycznych do określenia zderzenia między punktami jednego trójkąta a płaszczyzną innego trójkąta, a następnie określenia, czy te kolidujące punkty znajdują się wewnątrz drugiego trójkąta.

Wykrywanie zderzeń na podstawie kul otaczających obiekty

Wykrywanie zderzeń najlepiej przeprowadzać w hierarchicznych krokach: dla kul otaczających obiekty, następnie dla kul otaczających wieloboki, a na końcu wykorzystać przenikanie trójkątów. Liczenie kul otaczających obiekty jest bardzo proste; musisz znaleźć środek obiektu, następnie policzyć maksymalną odległość między środkiem a wierzchołkiem obiektu. Przechowując promień każdej otaczającej kuli, możesz przeprowadzać test zderzenia, dodając promienie i sprawdzając je z odległością środków obiektów. Jeśli suma jest większa niż odległość środków, kule nie przenikają się.

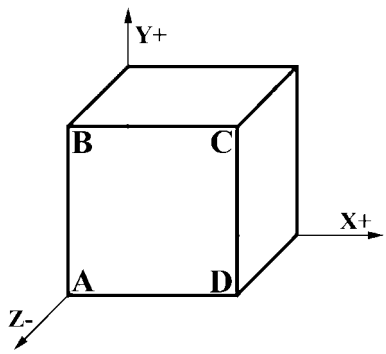
Przejdźmy przez to krok po kroku. Najpierw musimy określić środek siatki. Jedna z metod używa prostopadłościanu otaczającego obiekt i liczy środek przekątnej tej bryły (rysunek 4.5.1). Aby określić prostopadłościan, musimy poznać maksymalne i minimalne wartości *x*, *y* i *z* dla całego obiektu. Można to wykonać, przechodząc przez wszystkie wierzchołki i sprawdzając „aktualne" maksimum i minimum. Po sprawdzeniu wszystkich wierzchołków otrzymamy minimalny prostopadłościan, który otoczy obiekt.

Rysunek 4.5.1.
Znajdowanie środka



Dla prostopadłościanu z ośmioma punktami (**ABCDEFGH**, patrz rysunek 4.5.2) przypomnijmy sobie ułożenie wierzchołków

Rysunek 4.5.2.
Tworzenie prostopadłościanu otaczającego obiekt




```
A = (minx, miny, minz)
B = (minx, maxy, minz)
C = (maxx, maxy, minz)
D = (maxx, miny, minz)
E = (minx, miny, maxz)
```

```
F = (minx, maxy, maxz)
G = (maxx, maxy, maxz)
H = (maxx, miny, maxz)
```

Teraz znajdź środek, uśredniając maksymalny i minimalny punkt prostopadłościanu (w naszym przypadku są to punkty **A** i **G**).

```
// Wzór na środek: dane A(x1,y1,z1) i B(x2,y2,z2)
// Środek linii między punktami A i B jest następujący
// [(x1+x2)/2, (y1+y2)/2, (z1+z2)/2]
środek.x = (A.x + G.x)/2;
środek.y = (A.y + G.y)/2;
środek.z = (A.z + G.z)/2;
```

Promień otaczającej obiekt kuli można łatwo obliczyć, przechodząc przez kolejne wierzchołki i znajdując odległość między wierzchołkiem i środkiem obiektu. Jeśli uzyskana odległość jest większa od aktualnego maksimum, zastępujemy maksimum uzyskaną wartością. Po sprawdzeniu wszystkich wierzchołków maksymalna odległość jest promieniem kuli (oczywiście dosyć naturalną optymalizacją jest liczenie pierwiastka kwadratowego dopiero na końcu).

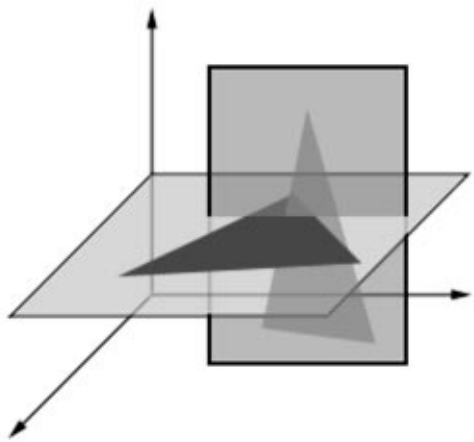
```
// Wzór na odległość:
// odl = sqrt[((x2-x1)^2) + ((y2-y1)^2) + ((z2-z1)^2)]
// odlkw = ((x2-x1)^2) + ((y2-y1)^2) + ((z2-z1)^2)
dla każdego wierzchołka v w obiekcie {
  aktualna_odległość_kw = odlkw(obiekt.środek, v);
  if (aktualna_odległość_kw > max_odległość_kw)
    max_odległość_kw = aktualna_odległość_kw;
}
obiekt.promień = sqrt(max_odległość_kw);
```

Całość powtórzymy na poziomie wielokątów; sprawdzanie przy użyciu otaczających kul jest proste i szybkie, więc użycie tej metody przy tym teście wydaje się logiczne. Po wygenerowaniu otaczających prostopadłościanów i kul dla każdego obiektu i wieloboku możemy zająć się najciekawszą częścią tego rozdziału: testem przecinania się trójkątów! Wyjmij książkę z geometrii — może Ci się przydać.

Wykrywanie zderzeń trójkątów

Przedstawiana metoda wykrywania zderzeń dwóch trójkątów nie jest trudna do zrozumienia, ale wykorzystuje pewną matematyczną sztuczkę. Wyobraź sobie, że mamy dwa trójkąty w trójwymiarowej przestrzeni (patrz rysunek 4.5.3). O obydwu musimy zebrać pewną ilość informacji. Dla jednego z nich musimy wyznaczyć równanie płaszczyzny, na której się znajduje. Jeśli masz dobrą pamięć, to zapewne wiesz, że ma ono postać: $Ax + By + Cz + D = 0$. A, B, C i D określimy, licząc iloczyn wektorowy wierzchołków.

```
// dany jest trójkąt tri1 z wierzchołkami a, b i c
// vector3 a, b, c;
vector3 v1, v2, cross_v1xv2;

// tworzymy wektory v1 i v2
// (tri1.b - tri1.a i tri1.c - tri1.a)
v1 = tri1.b - tri1.a;
v2 = tri1.c - tri1.a;
```



Rysunek 4.5.3.
Dwa przecinające się trójkąty

```
// UWAGA: możesz pominąć ten krok i użyć własnych normalnych płaszczyzny, jeśli
// gdzieś je przechowujesz. Uzyskujemy iloczyn wektorowy v1 i v2 (wektor)
cross_v1xv2 = CrossProduct(v1, v2);

// Wstawiamy te wartości do Ax+By+Cz+D=0
tri1.pA = cross_v1xv2.x;
tri1.pB = cross_v1xv2.y;
tri1.pC = cross_v1xv2.z;

// Korzystamy z równania Ax+By+Cz+d=0, by obliczyć D
// Jeśli punkt P(x0,y0,z0) jest punktem na wieloboku
// A = cross_v1xv2.x
// B = cross_v1xv2.y
// C = cross_v1xv2.z
// D = (-A*x0-B*y0-C*z0)
tri1.pD = -DotProduct(cross_v1xv2, P);
```

Przecięcie linia-płaszczyzna

Mamy już równanie płaszczyzny pierwszego trójkąta (`tri1`) i możemy przejść do następnego etapu: sprawdzania, czy drugi trójkąt (`tri2`) koliduje z płaszczyzną `tri1`. Jest to wykonywane w kilku krokach. Główna idea opiera się na tym, że między dwoma wierzchołkami `tri2` prowadzimy linię i sprawdzamy, czy przecina ona płaszczyznę `tri1`. Jeśli punkt kolizji znajduje się między tymi dwoma wierzchołkami, `tri2` przecina płaszczyznę `tri1`; jeśli nie znajduje się miedzy wierzchołkami, przechodzimy do następnych dwóch linii tworzących `tri2`, aby sprawdzić, czy przypadkiem tam nie występuje kolizja.

Przecięcie linia-płaszczyzna rozwiązujemy, używając równań parametrycznych. Dla danych wierzchołków **a**(*x0*,*y0*,*z0*) i **b**(*x1*,*y1*,*z1*), tworzymy równanie **a**(*x0*,*y0*,*z0*) ∗ *t* = **b**(*x1*,*y1*,*z1*) ∗ (1 – *t*), gdzie *t* to współczynnik interpolacji zmieniający się od 0 do 1. Gdy *t*=0 jesteś w punkcie **b**, gdy jest równe 1 — w **a**. Jeśli wstawimy te równania parametryczne do równania płaszczyzny, będziemy mogli policzyć *t*.

$$A*(x0*t + x1*(t-1)) + B*(y0*t + y1*(1-t)) + C*(z0*t + z1*(1-t)) + D = 0$$

Po przekształceniach otrzymamy:

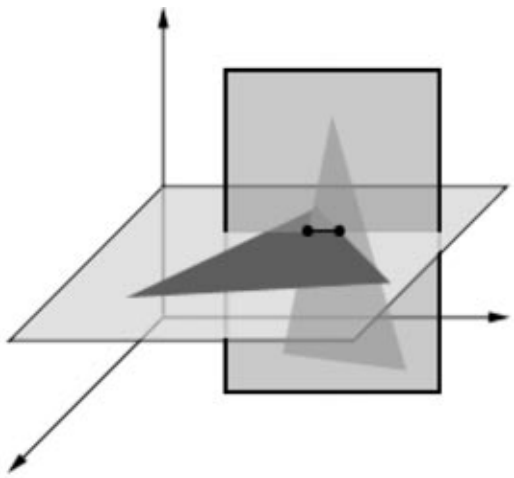
$$t = -(A*x1 + B*y1 + C*z1 + D)/(A*(x0 - x1) + B*(y0 - y1) + C*(z0 - z1))$$

Oto kod obliczający *t*:

```
// i0 = (A*x0) + (B*y0) + (C*z0)
i0 = (tri1->pA*a->x) + (tri1->pB*a->y) + (tri1->pC*a->z);

// i1 = (A*x1) + (B*y1) + (C*z1)
i1 = (tri1->pA*b->x) + (tri1->pB*b->y) + (tri1->pC*b->z);

// Tutaj uważaj na możliwe dzielenia przez zero (np. gdy i0 = i1)
final_t = -(i1 + tri1->pD) / (i0-i1);

// Teraz wstaw final_t do funkcji x(), y() i z(), aby otrzymać punkt przecięcia
final_x = (((a->x)*(final_t))+((b->x)*(1-final_t)));
final_y = (((a->y)*(final_t))+((b->y)*(1-final_t)));
final_z = (((a->z)*(final_t))+((b->z)*(1-final_t)));
```

W ten sposób otrzymujemy punkt, w którym linia przecina płaszczyznę (rysunek 4.5.4). Oczywiście wartość *t*, którą obliczymy, musi się znajdować w przedziale 0 – 1; w przeciwnym przypadku przecięcie nie znajduje się między wierzchołkami! Szczególny przypadek, na który musisz zwrócić uwagę w tym kroku, to wystąpienie pionowych linii. Najszybszą metodą określenia przecięcia linii jest wstawienie *x* i *z* dla obydwu punktów (**a** i **b**) do równania płaszczyzny trójkąta i obliczenie *y*. Wtedy punktem przecięcia będzie punkt (**a**.*x*, **obliczony**.*y*, **a**.*z*).

Rysunek 4.5.4. *Określanie punktu przecięcia*



Spłaszczanie trójkąta

Będziemy teraz używać praworęcznego systemu współrzędnych. Wyobraź sobie spłaszczenie trójkąta względem jednej z płaszczyzn współrzędnych, zależnie od obrotu trójkąta. Może on na przykład utracić współrzędną *y* i zachować współrzędne *x* i *z*. Dokładniej — ta idea nie oznacza wyrzucenia współrzędnej *y*; wyrzucamy odpowiednią współrzędną, aby uzyskać efekt spłaszczenia. Wyboru najłatwiej dokonać, przypatrując się nor-

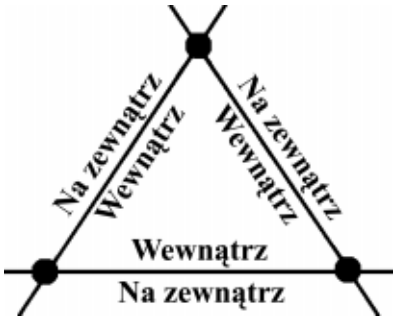
malnej płaszczyzny. Jeśli ustalisz, który element ma największą wartość bezwzględną, możesz znaleźć płaszczyznę, względem której możesz spłaszczyć, aby trójkąt nie stał się linią prostą (na przykład gdy pionowy trójkąt utraci współrzędną *y*). Na przykład jeśli element *x* jest największy, będziesz rzutował na płaszczyznę *yz*. Niezależnie od obrotu, uzyskany trójkąt będzie płaski (jakby leżał na stole — rysunek 4.5.5). Ta technika jest bardzo przydatna, ponieważ możemy wtedy bardzo łatwo stwierdzić, czy uzyskany punkt przecięcia (po spłaszczeniu w podobny sposób), znajduje się wewnątrz trójkąta. Kod z wydruku 4.5.1 spłaszcza trójkąt względem jednej z płaszczyzn współrzędnych.

Rysunek 4.5.5. *Rzutowanie wierzchołków*

Testowanie, czy punkt jest wewnątrz trójkąta

Teraz, kiedy już spłaszczyliśmy współrzędne, potrzebujemy pewnych obliczeń matematycznych, by określić, czy punkt przecięcia znajduje się wewnątrz spłaszczonego trójkąta, czy też nie. Test można przeprowadzić na kilka sposobów: my zamierzamy skorzystać z równania dla każdej linii spłaszczonego trójkąta. Zauważ, że niezależnie od tego, na jaką płaszczyznę rzutowaliśmy, spłaszczone punkty będziemy określać współrzędnymi *x* i *y*. Robimy tak dlatego, że w zasadzie zredukowaliśmy problem do dwóch wymiarów, czyli osi X i Y. Najpierw musimy znaleźć punkt, który *na pewno* znajduje się wewnątrz trójkąta. Najłatwiej zrobić to, obliczając środek trójkąta, liczony jako średnia wierzchołków: ((*x0*+*x1*+*x2*)/2, (*y0*+*y1*+*y2*)/2). Skoro znamy już kierunek do wnętrza trójkąta, możemy sprawdzić, czy nasz punkt znajduje się po „wewnętrznej" stronie linii tworzonych przez wierzchołki (rysunek 4.5.6).

Dla danych wierzchołków **v0** i **v1** najpierw znajdujemy równanie linii przechodzącej przez te punkty w postaci *y*=*mx*+*b*. Zapewne pamiętasz, że wzór na nachylenie (*m*) to (*y1* – *y0*)/(*x1* – *x0*), więc możemy znaleźć *b*, używając wcześniej obliczonego nachylenia i punktu, przez który przechodzi prosta. Skoro mamy już równanie linii w postaci z nachyleniem, możemy sprawdzić, czy punkt przecięcia leży wewnątrz spłaszczonego trójkąta. Robimy to, porównując wartości *y*. Jeśli wstawisz wartość *x* spłaszczonego punku przecięcia do wzoru *y*=*mx*+*b*, otrzymasz wartość *y* w punkcie *x*. Teraz sprawdzisz, czy obliczony środek trójkąta znajduje się nad, czy poniżej linii, testując wartość *y*. Wiemy, że znajduje się w środku, więc nasz punkt przecięcia musi mieć wartość *y* w tym samym



Rysunek 4.5.6. *Określanie obszaru zajmowanego przez trójkąt*

kierunku co środek trójkąta. Jeśli tak rzeczywiście jest, punkt znajduje się wewnątrz trójkąta względem linii **ab**. Musimy to jeszcze powtórzyć dla linii **bc** i **ca**. Jeśli w każdym teście dowiedzieliśmy się, że punkt znajduje się w środku trójkąta, rzeczywiście tak jest. Należy jednak brać pod uwagę szczególny przypadek: rzutowanie pionowej linii. Za pomocą $y=mx+b$ nie możesz narysować takiej linii. Jeśli wystąpi taka sytuacja, sprawdzasz współrzędną x zamiast y; to znaczy, że najpierw określasz, po której stronie pionowej linii znajduje się wewnętrzny punkt. Następnie sprawdzasz wartość x rzutowanego punktu przecięcia. Jeśli jest po tej samej stronie, co poprzedni punkt, znajduje się wewnątrz trójkąta względem sprawdzanej linii. Implementacje znajdziesz na wydruku 4.5.2.

Sprawdzaj wszystkie linie w obydwu trójkątach!

Oczywiście jeśli pierwsza linia nie przecina płaszczyzny, musimy sprawdzić pozostałe. Wystarczy znaleźć jedną kolizję linia-trójkąt, aby wykazać, że trójkąty nachodzą na siebie. Resztę przykładowego kodu znajdziesz na wydruku 4.5.3.

Jeśli nie wykryłeś żadnego zderzenia, musisz odwrócić procedurę, zaczynając tym razem od drugiego trójkąta. Tym samym upewniasz się, że na pewno wykryjesz zderzenie.

Wydruk 4.5.1

```
if (x==FALSE) { // opuszczam współrzędną x
    a1 = tri->a.y;
    b1 = tri->a.z;
    a2 = tri->b.y;
    b2 = tri->b.z;
    a3 = tri->c.y;
    b3 = tri->c.z;
    a4 = vert->y;
    b4 = vert->z;
    inside = 0;
  }
  else if (y==FALSE) { // opuszczam współrzędną y
    a1 = tri->a.x;
    b1 = tri->a.z;
    a2 = tri->b.x;
    b2 = tri->b.z;
    a3 = tri->c.x;
    b3 = tri->c.z;
    a4 = vert->x;
    b4 = vert->z;
    inside = 0;
```

```
  }
  else if (z==FALSE) { // opuszczam współrzędną z
    a1 = tri->a.x;
    b1 = tri->a.y;
    a2 = tri->b.x;
    b2 = tri->b.y;
    a3 = tri->c.x;
    b3 = tri->c.y;
    a4 = vert->x;
    b4 = vert->y;
    inside = 0;
  }
```

Wydruk 4.5.2

```
// Poniższy kod sprawdza pionowe linie w spłaszczonym trójkącie.
// Nie można narysować linii za pomocą y=mx+b, więc musimy sprawdzać,
// czy spłaszczony punkt przecięcia znajduje się między współrzędną x
// dowolnej pionowej linii a środkiem trójkąta, aby przekonać się, czy
// punkt przecięcia znajduje się wewnątrz trójkąta względem pionowej linii.
AB_vert = BC_vert = CA_vert = FALSE;

// y=mx+b dla trzech linii tworzących trójkąt
if ((a2-a1)!=0) {
  m1 = (b2-b1)/(a2-a1); // a->b
  bb1 = (b1)-(m1*a1);    // y/(mx) przy użyciu wierzchołka a
} else if ((a2-a1)==0) {
  AB_vert = TRUE;
}

if ((a3-a2)!=0) {
  m2 = (b3-b2)/(a3-a2); // b->c
  bb2 = (b2)-(m2*a2);    // y/(mx) przy użyciu wierzchołka b
} else if ((a3-a2)==0) {
  BC_vert = TRUE;
}

if ((a1-a3)!=0) {
  m3 = (b1-b3)/(a1-a3); // c->a
  bb3 = (b3)-(m3*a3);    // y/(mx) przy użyciu wierzchołka c
} else if ((a1-a3)==0) {
  CA_vert = TRUE;
}

// znajdź średni punkt trójkąta (na pewno znajduje się on w jego wnętrzu)
center_x = (a1+a2+a3)/3;
center_y = (b1+b2+b3)/3;

// Sprawdź, czy (center_x,center_y) jest powyżej, czy poniżej linii,
// ustaw kierunek na UP, gdy jest wyżej i DOWN, gdy jest poniżej linii

// a->b
if (((m1*center_x)+bb1) >= center_y)
  DIRECTION(direction,UP);
else
  DIRECTION(direction,DOWN);
if (AB_vert==TRUE) {
```

```
  if ((a1<a4)&&(a1<center_x)) // rzutowana pionowo linia
    inside++;
  else if ((a1>a4)&&(a1>center_x)) // rzutowana pionowo linia
    inside++;
} else {
  if (direction==UP) {
    if (b4 <= ((m1*a4)+bb1)) // b4 mniejsze od y, aby był wewnątrz
      inside++;              // (linia powyżej punktu)
  } else if (direction==DOWN) {
    if (b4 >= ((m1*a4)+bb1)) // b4 większa od y, by był wewnątrz
      inside++;              // (linia poniżej punktu)
  }
}

// b->c
if (((m2*center_x)+bb2) >= center_y)
  DIRECTION(direction,UP);
else
  DIRECTION(direction,DOWN);
if (BC_vert==TRUE) {
  if ((a2<a4)&&(a2<center_x)) // rzutowana pionowo linia
    inside++;
  else if ((a2>a4)&&(a2>center_x)) // rzutowana pionowo linia
    inside++;
} else {
  if (direction==UP) {
    if (b4 <= ((m2*a4)+bb2)) // b4 mniejsze od y, aby był wewnątrz
      inside++;              // (linia powyżej punktu)
  } else if (direction==DOWN) {
    if (b4 >= ((m2*a4)+bb2)) // b4 większe od y, aby był wewnątrz
      inside++;              // (linia poniżej punktu)
  }
}
// c->a
if (((m3*center_x)+bb3) >= center_y)
  DIRECTION(direction,UP);
else
  DIRECTION(direction,DOWN);
if (CA_vert==TRUE) {
  if ((a3<a4)&&(a3<center_x)) // rzutowana pionowo linia
    inside++;
  else if ((a3>a4)&&(a3>center_x)) // rzutowana pionowo linia
    inside++;
} else {
  if (direction==UP) {
    if (b4 <= ((m3*a4)+bb3)) // b4 mniejsze od y, aby był wewnątrz
      inside++;              // (linia powyżej punktu)
  } else if (direction==DOWN) {
    if (b4 >= ((m3*a4)+bb3)) // b4 większe od y, aby był wewnątrz
      inside++;              // (linia poniżej punktu)
  }
}
if (inside==3) {
  return TRUE;
} else {
  return FALSE;
}
```

Wydruk 4.5.3

```
// Przejdź przez wszystkie trzy linie tworzące trójkąt
// Pierwsza iteracja  (a,b)
p=line_plane_collision((vertex_ptr)&tri2.a,(vertex_ptr)&tri2.b,(triangle_ptr)&tri1);

// Określ, na którą oś rzutować
// X jest największe
if ((abs(tri1.pA)>=abs(tri1.pB))&&(abs(tri1.pA)>=abs(tri1.pC)))
  temp = point_inside_triangle((triangle_ptr)&tri1,(vertex_ptr)&p,
                                                    FALSE,TRUE,TRUE);
// Y jest największe
else if ((abs(tri1.pB)>=abs(tri1.pA))&&(abs(tri1.pB)>=abs(tri1.pC)))
  temp = point_inside_triangle((triangle_ptr)&tri1,(vertex_ptr)&p,
                                                    TRUE,FALSE,TRUE);
// Z jest największe
else if ((abs(tri1.pC)>=abs(tri1.pA))&&(abs(tri1.pC)>=abs(tri1.pB)))
  temp = point_inside_triangle((triangle_ptr)&tri1,(vertex_ptr)&p,
                                                    TRUE,TRUE,FALSE);

if (temp==TRUE) {
  // Sprawdzamy punkt, by sprawdzić, czy leży między wierzchołkami
  // Najpierw sprawdzaj szczególny przypadek z pionowymi liniami
  if ((tri2.a.x == tri2.b.x)&&(tri2.a.z == tri2.b.z)) {
    if (((tri2.a.y <= p.y)&&(p.y <= tri2.b.y))||
        ((tri2.b.y <= p.y)&&(p.y <= tri2.a.y)))
      return TRUE;
  }
  // Koniec sprawdzania szczególnego przypadku

  // Teraz sprawdź punkt względem linii
  if (point_inbetween_vertices((vertex_ptr)&tri2.a,(vertex_ptr)&tri2.b,
                                          (triangle_ptr)&tri1)==TRUE)
    return TRUE;
  else
    return FALSE;
}

// Druga iteracja (b,c)
p=line_plane_collision((vertex_ptr)&tri2.b,(vertex_ptr)&tri2.c,
                                        (triangle_ptr)&tri1);

// Określ, na którą oś rzutować
// X jest największe
if ((abs(tri1.pA)>=abs(tri1.pB))&&(abs(tri1.pA)>=abs(tri1.pC)))
  temp = point_inside_triangle((triangle_ptr)&tri1,(vertex_ptr)&p,
                                                    FALSE,TRUE,TRUE);
// Y jest największe
else if ((abs(tri1.pB)>=abs(tri1.pA))&&(abs(tri1.pB)>=abs(tri1.pC)))
  temp = point_inside_triangle((triangle_ptr)&tri1,(vertex_ptr)&p,
                                                    TRUE,FALSE,TRUE);
// Z jest największe
else if ((abs(tri1.pC)>=abs(tri1.pA))&&(abs(tri1.pC)>=abs(tri1.pB)))
  temp = point_inside_triangle((triangle_ptr)&tri1,(vertex_ptr)&p,
                                                    TRUE,TRUE,FALSE);

if (temp==TRUE) {
  // Sprawdzamy punkt, by sprawdzić, czy leży między wierzchołkami
```

```
  // Najpierw sprawdzaj szczególny przypadek z pionowymi liniami
  if ((tri2.b.x == tri2.c.x)&&(tri2.b.z == tri2.c.z)) {
    if (((tri2.b.y <= p.y)&&(p.y <= tri2.c.y))||
        ((tri2.c.y <= p.y)&&(p.y <= tri2.b.y)))
      return TRUE;
  }

  // Teraz sprawdź punkt względem linii
  if (point_inbetween_vertices((vertex_ptr)&tri2.b,(vertex_ptr)&tri2.c,
                                          (triangle_ptr)&tri1)==TRUE)
    return TRUE;
  else
    return FALSE;
}

// Trzecia iteracja  (c,a)
p=line_plane_collision((vertex_ptr)&tri2.c,(vertex_ptr)&tri2.a,
                                         (triangle_ptr)&tri1);

// Określ, na którą oś rzutować
// X jest największe
if ((abs(tri1.pA)>=abs(tri1.pB))&&(abs(tri1.pA)>=abs(tri1.pC)))
  temp = point_inside_triangle((triangle_ptr)&tri1,(vertex_ptr)&p,
                                                  FALSE,TRUE,TRUE);
// Y jest największe
else if ((abs(tri1.pB)>=abs(tri1.pA))&&(abs(tri1.pB)>=abs(tri1.pC)))
  temp = point_inside_triangle((triangle_ptr)&tri1,(vertex_ptr)&p,
                                                  TRUE,FALSE,TRUE);
// Z jest największe
else if ((abs(tri1.pC)>=abs(tri1.pA))&&(abs(tri1.pC)>=abs(tri1.pB)))
  temp = point_inside_triangle((triangle_ptr)&tri1,(vertex_ptr)&p,
                                                  TRUE,TRUE,FALSE);

if (temp==TRUE) {
  // Sprawdzamy punkt, by sprawdzić, czy leży między wierzchołkami
  // Najpierw sprawdzaj szczególny przypadek z pionowymi liniami
  if ((tri2.c.x == tri2.a.x)&&(tri2.c.z == tri2.a.z)) {
    if (((tri2.c.y <= p.y)&&(p.y <= tri2.a.y))||
        ((tri2.a.y <= p.y)&&(p.y <= tri2.c.y)))
      return TRUE;
  }

  // Teraz sprawdź punkt względem linii
  if (point_inbetween_vertices((vertex_ptr)&tri2.c,(vertex_ptr)&tri2.a,
                                          (triangle_ptr)&tri1)==TRUE)
    return TRUE; // Punkt przecięcia znajduje się wewnątrz trójkąta lub na linii
  else
    return FALSE;
}
return FALSE; // Domyślna wartość - brak zderzenia
```